CacheQuery: A Practical Asymmetric Communication Algorithm

Yu-Sian Li[1], Cao Minh Trang[2], Xin Huang[3], Cheng-Hsin Hsu[1], and Po-Ching Lin[4]
[1]Department of Computer Science, National Tsing Hua University, Hsin Chu, Taiwan
[2]NETS Research Group, Universitat Pompeu Fabra (UPF), Barcelona, Spain
[3]Deutsche Telekom R&D Lab USA, 5050 El Camino Real 221, Los Altos, CA 94022
[4]Department of Computer Science and Information Engineering, National Chung Cheng University, Chia Yi, Taiwan

***Abstract*—We consider the problem of asymmetric communications, which are common in many access networks. We propose a new asymmetric communication algorithm, called CacheQuery to leverage on the already deployed downlink bandwidth and receiver capability to accelerate the uplink data transfer from one or multiple senders to a receiver. The design of CacheQuery differs from all previous asymmetric communication algorithms in two ways: (i) CacheQuery supports more flexible matching mechanism to identify redundant packet payload and (ii) CacheQuery allocates a small sender cache to absorb the potentially high downlink traffic overhead incurred by asymmetric communications. The trace-driven simulations indicate that, compared to existing asymmetric communication algorithms, CacheQuery achieves higher uplink transfer speed, yet reduces downlink traffic overhead.**

***Index Terms*—Asymmetric communications, bandwidth asymmetry, capability asymmetry, redundancy elimination**

I. Introduction

Network communications are often constrained by asymmetric resources at the sender and receiver in terms of network *bandwidth* and end-device *capability*. Bandwidth asymmetry, illustrated in Fig. 1 is common in various access networks, including Asymmetric Digital Subscriber Lines (ADSLs), cable modems, 3G/4G cellular networks, and hybrid satellite-terrestrial access [1]. For bandwidth asymmetric channels, the downlink bandwidth could be up to 1000 times higher than the uplink bandwidth [2]–[4], due to business concerns and technology limitations. Sending large files over these channels results in long upload time and degraded user experience. However, increasing uplink bandwidth of these channels, such as upgrading to premium Internet plans or deploying additional network infrastructure, is quite costly.

Some end-devices, such as smartphones and sensors, have limited memory size, processing power, and battery capacity. These end-devices are not capable to run computational- and storage-intensive algorithms, and are often connected to powerful remote servers. This is referred to as capability asymmetry. Recently, the negative impacts of capability asymmetry are gradually surfacing, for example: (i) more smartphone applications push computations into clouds, which may dramatically increase the network traffic [5], and (ii) the Internet of Things (IoT) paradigm connects a huge number of sensors to the public Internet, which imposes a tremendous amount of traffic [6], [7]. Although upgrading these end-devices, e.g., to perform real-time data compression, may reduce the network traffic, doing so will render many mobile and IoT applications less commercially-viable.

Y. Li and C. Hsu are partially supported by the National Science Council (NSC) of Taiwan, grant # 100-2218-E-007-015-MY2.

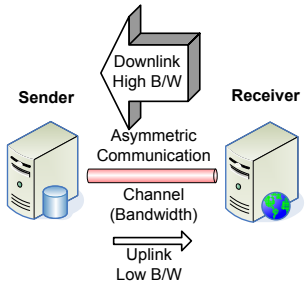


Fig. 1. A bandwidth asymmetric channel.

We collectively call communications over bandwidth and capability asymmetric channels as *asymmetric communications*. In this paper, we study the problem of increasing uplink goodput in asymmetric communications by capitalizing the otherwise wasted downlink bandwidth and receiver capability. We define *goodput* as the effective application-level throughput, excluding all protocol and error-recovery overhead. In particular, we design a new asymmetric communication algorithm, called CacheQuery, on top of TCP to increase the uplink goodput from one or multiple senders to one receiver. We define goodput gain of an asymmetric communication algorithm as the relative goodput improvement compared to the standard TCP transfer. The proposed CacheQuery algorithm aims to maximize the uplink goodput gain.

We note that CacheQuery is not the first proposal for increasing uplink goodput gain. In particular, we implemented several existing asymmetric communication algorithms [8]–[12] in our earlier work [13], and we evaluated them using synthetic traces. In this paper, we conduct more simulations using *real* network traces, which reveals the inefficiency of the existing asymmetric communication algorithms and the superior performance achieved by the proposed CacheQuery algorithm.

This paper makes the following main contributions:

- We quantify the limitations of existing asymmetric communication algorithms in Sec. III. Via trace-driven simulations, we show that the existing algorithms only achieve

limited uplink goodput gain: as low as 2%.

- We propose CacheQuery, which is a new asymmetric communication algorithm, in Sec. IV. CacheQuery supports flexible packet matching for higher uplink goodput gain and employs a small cache at the sender to mitigate the potential issue of high downlink traffic amount in existing algorithms.
- We conduct extensive trace-driven simulations to evaluate CacheQuery in Sec. V. Our evaluation results show that CacheQuery outperforms existing asymmetric communication algorithms by far: up to 50 times improvement on uplink goodput gain and up to 384 times reduction on downlink traffic amount are observed.

II. Related Work

A. Asymmetric Communication Algorithms

The existing asymmetric communication algorithms can be categorized into two classes: static and dynamic. The static asymmetric communication problem is first considered by Adler and Maggs [8]. The problem is static because it assumes the communication packets follow a known and fixed probability distribution. The static problem is also considered in other work [9], [10], and we refer to those algorithms as static algorithms. It is reported that the static problem is built on top of a strong assumption that the receiver knows the probability distribution of packets, which is unrealistic [11], [12]. Gagie therefore considers a dynamic problem, in which multiple clients send packets to a server, following a probability distribution that is unknown to both senders and the receiver [11], [12]. Gagie proposes algorithms for the dynamic problem [11]. We refer to these algorithms as dynamic algorithms. We do not assume a receiver knows the probability distribution of the packets, and thus we only consider dynamic algorithms throughout this paper.

We present the main idea behind dynamic algorithms below. Generally, each asymmetric communication algorithm maintains a *cache* at the receiver, to keep track of t *seen* packets sent from one or more senders, where t is a system parameter. The receiver uses this cache in the following way. For each incoming packet the receiver *guesses* the packet according to the cache, and *asks* the sender if the guess is correct. The sender either: (i) confirms the correctness of the guess and moves on to the next packet or (ii) sends the receiver a *hint* to adjust its guess on the same packet in the next round. Each packet is delivered in multiple rounds $r \geq 1$. A receiver updates its cache once successfully receiving a packet, in order to leverage on the known packet pattern for fewer guess rounds. Since the packet distribution is unknown and dynamic, the receiver may saturate all educational guesses, and has to ask the sender to transmit the packet as-is. A sender also sends a packet as-is if the number of rounds r exceeds an algorithm-specific threshold $r_{\max}$, in order to avoid long latency.

There are four asymmetric communication algorithms proposed in the literature [11], [12]: Dynamic Bit-Efficient-Split (DBES), TreeQuery, ListQuery, and QueueQuery. Due to the space limitations, interested readers are referred to our technical report [14] for details of these algorithms. In our previous work [13], we implemented DBES and ListQuery algorithms in a packet level simulator. In this work, we also implement QueueQuery for comparisons. In Sec. III, we conduct trace-driven simulations to quantify the potential of DBES, ListQuery, and QueueQuery using real network traces.

TABLE I
Packet traces from real servers

Trace	Server Type	Location	Duration (hr)	Size (MB)
T1	Enterprise Server	US East Coast	168	59
T2	Enterprise Server	US West Coast	98	153
T3	Home Server	Taiwan	60	404
T4	Home Server	US West Coast	122	821
T5	University Server	Canada West Coast	47	12,568

B. Redundancy Elimination Algorithms

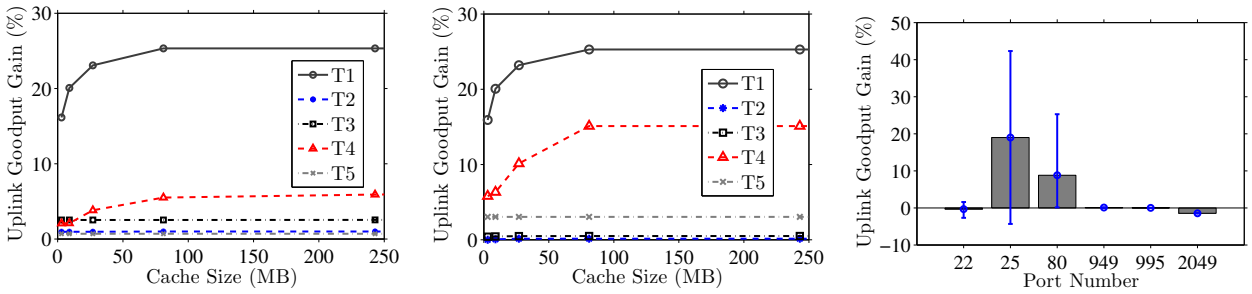
There are several redundancy elimination algorithms proposed in the literature, which can also be used to increase uplink goodput of asymmetric communications to some degree. Online compression algorithms [15] compress the data payload in real time and have been deployed in commercial routers [16]. Protocol-independent redundancy elimination algorithms [17], on the other hand, remove duplicated packets. In contrast to CacheQuery, while redundancy elimination algorithms [15]–[17] may reduce the uplink traffic amount, they cannot leverage on downlink bandwidth and receiver capability for faster upload speed. Moreover, existing redundancy elimination algorithms demand considerable resources, including memory, CPU cycles, and energy at senders, and thus are not suitable when the senders have limited resources. Last, unlike the proposed CacheQuery, they cannot leverage redundancy across multiple senders.

Recently, Zohar et al. [18] propose a receiver-driven redundancy elimination algorithm, called PACK, to minimize the cost of cloud customers at the expense of potentially overloading mobile computers and sensors that are downloading from the cloud. PACK is more suitable to larger files such as video and email attachments, and the authors of [18] recommend to fall back to sender-driven (traditional) redundancy elimination algorithms, such as [17], for smaller files. In fact, even for video files, PACK leads to slightly lower goodput gain compared to sender-driven redundancy elimination algorithms, according to their evaluation results [18]. The lower goodput gains can be attributed to PACK's cache-less sender design. In contrast, our proposed CacheQuery algorithm employs a small sender cache for higher goodput gains. In Sec. V, we will show that the proposed CacheQuery algorithm outperforms a state-of-the-art sender-driven redundancy elimination algorithm [17] in terms of the goodput gain. This in turn implies that CacheQuery outperforms PACK as well.

III. Limitations of Current Asymmetric Communication Algorithms

A. Potential of Asymmetric Communication Algorithms

We collected egress packet traces from five real servers in enterprise, home, and university networks using `tcpdump`.



(a) (b) (c)

Fig. 2. Goodput gains achieved by ListQuery, results from: (a) protocol-independent cache, (b) HTTP cache, and (c) per-protocol caches with 1 GB cache.

All the servers ran Linux, and had 4-12 local users. We collected the traces without asking users to change their daily usage patterns. Some services, e.g., Web services, may have many more anonymous remote users. Table I summarizes the information of individual traces. The trace files enable us to perform realistic trace-driven simulations. We implemented the DBES, ListQuery, and QueueQuery algorithms in the NS-2 simulator. We however found that conducting NS-2 simulations is quite time consuming. Therefore, we also developed our own event-driven simulator using C/C++, which runs more than 100 times faster than NS-2 when the network topology is simple. We ran several simulations using both NS-2 and our simulator, and carefully compared the simulation results to verify the correctness of our simulator. In the rest of this paper, we report the simulation results from our own simulator. We let the maximum packet size be 1500 bytes, and vary the cache size from 3 to 1500 MB. We set $k = 1$ for ListQuery and QueueQuery algorithms. We conduct two sets of simulations. First, we use each trace file to drive the simulator with one of the three considered algorithms. This is to emulate the scenarios where a protocol-independent cache is used between any pair of sender and receiver. Second, we split each trace file into smaller trace files based on their port numbers. We then run the simulators with the split trace files, so as to emulate the scenarios where a per-protocol cache is employed. We use the uplink goodput gain as performance metric. The goodput gain is defined as the relative goodput increase of an asymmetric communication algorithm compared to a standard TCP data stream.

The simulation results indicate that DBES never results in positive goodput gain. Furthermore, throughout our simulation, QueueQuery always achieves similar, but slightly worse uplink goodput gain compared to ListQuery. Therefore, we only report results from ListQuery. We found that the uplink goodput gain does not increase when cache size is larger than 250 MB; hence, we only plot the results with cache size in $[3, 250]$. We first present the results from ListQuery. We plot its protocol-independent uplink goodput gain in Fig. 2(a). This figure shows that only one trace (T1) results in uplink goodput gain higher than 7%; three traces (T2, T3, and T5) lead to negligible ($< 2\%$) uplink goodput gain. Next, we report the per-protocol uplink goodput of HTTP traffic in Fig. 2(b). Compared to Fig. 2(a), the uplink goodput gain of HTTP is generally higher. Nevertheless, majority of the traces (T2, T3, and T5) still lead to small ($< 3\%$) uplink goodput gain. Last, we compute the uplink per-protocol goodput gain of individual traces, and plot their mean, minimum, and maximum gains in Fig. 2(c). This figure shows that the uplink goodput gains are low, with exceptions of the HTTP (port 80) and SMTP (port 25) protocols. Even for HTTP and SMTP, the worst per-trace uplink goodput gains are $< 10\%$.

B. Discussion

We take a closer look at the packets in the traces to determine the root causes of the inferior performance of the current asymmetric communication algorithms. We found that these algorithms are limited in the sense that they only leverage the redundancy of *exact-match* packets. In actual traffic traces, however, exact-match packets do not occur too often. Rather, we often observe packets that are almost matching except a few *critical bytes* that are different from one another. Although there is a high redundancy between the two packets, the current algorithms will treat them as different packets. Furthermore, a common byte range may appear in different positions of two packets, which are then considered as different packets by the current algorithms. Take HTTP packets as examples, metadata such as timestamps, cookie IDs, and hit counts are critical bytes, which may have variable length. This in turn results in diverse *offsets*. We refer to packets that only differ by critical bytes and diverse offsets as *partial-match* packets. We believe that the current algorithms achieve low uplink goodput gains because they cannot identify the partial-matches.

IV. A New Asymmetric Communication Algorithm: CacheQuery

For the ease of presentation, we first describe the single-sender scenario. We will discuss the multi-sender extension in Sec. VI-A.

A. Overview

The main objective of CacheQuery is to maximize the uplink goodput gain by supporting partial-match, which allows us to capitalize common byte ranges with arbitrary offsets and lengths shared between the current and a historical packet. CacheQuery resides in between the transport and application layers, and provides a boosted uplink data transfer service to applications. CacheQuery can be deployed on two end-systems of asymmetric communications or on an in-network proxy.

More elaborated topologies are also possible. For example, two hosts of asymmetric communications may connect through a common proxy for goodput gains in *both* directions. Multiple proxies at different Internet Service Providers (ISPs) may also collaborate with each other by establishing high-bandwidth channels among them.

B. Packet Caches

Similar to previous asymmetric communication algorithms, CacheQuery maintains a cache of historical packets at the receiver. We let the receiver cache size be B_r MB, which is determined by the receiver's capability. CacheQuery also allocates a packet cache at the sender to speed up the process, and we let the sender cache size be B_s MB, which is determined by the sender capability. At the sender, each new packet is compared against the historical packets in the sender cache to find the longest common byte range. The matching byte ranges are encoded for reducing the uplink data redundancy.

In CacheQuery, the receiver cache cannot be smaller than the sender cache; otherwise some encoded byte ranges might not be decodable at the receiver. Therefore, we have $B_r \geq B_s$. Having a larger receiver cache makes sense for capability-constrained mobile and sensing devices, because the receiver can *help* the senders to memorize more historical packets for higher uplink goodput gain. More specifically, the receiver periodically transmits a subset of the receiver cache to the sender. The sender then uses this cache subset to replace the old, potentially outdated, sender cache. This is referred to as *cache update*. The receiver also keeps a copy of the sender cache for the decoding purpose, which is called sender cache *shadow* in CacheQuery.

The cache update is performed once every f packets, where *update frequency* f is a system parameter. The update frequency controls the tradeoff between downlink traffic amount and uplink goodput gain, because less frequent updates result in more outdated sender cache, but save some downlink traffic. We will quantify this tradeoff via trace-driven simulations in Sec. V.

In CacheQuery, we assume the size of each cache update is B_s for simplicity. That is, the receiver always *fills up* the entire sender cache in each cache update. Given that $B_r \geq B_s$, CacheQuery has to define a *selection policy* to maximize the chance of identifying common byte ranges at the sender for higher uplink goodput gain. Typical selection policies include: (i) Most-Recently-Used (MRU) and (ii) Most-Frequently-Used (MFU) packets. We consider a general *hybrid* policy P_β ($0 \leq \beta \leq 1$), which selects β MRU and $1-\beta$ MFU packets. It is clear that P_β covers the full spectrum of selection policies between (and including) MRU and MFU. Upon a common byte range matches a packet, CacheQuery increases its hit count by one and/or updates its last-seen timestamp. In Sec. V, we will empirically study the optimum value of the parameter β in order to maximize the uplink goodput gain. Different from f, β does not affect the downlink traffic amount, yet has a potential to improve the uplink goodput gain.

C. Efficient Partial-Match Algorithm

Selecting representative windows. To avoid excessive computational complexity at the receiver, each packet is scanned and marked with one or multiple *representative* windows, where each window is w-bytes long. We refer to w as the window size. The partial-match process uses these representative windows as *entering* points to locate matching byte ranges and thus the complexity can be controlled. Moreover, we use a window sampling frequency p to throttle the number of representative windows. In particular, CacheQuery only considers $1/p$ qualified representative windows for the sake of lower computational complexity. w and p are system parameters, and could affect the performance of CacheQuery. We empirically compared several w and p values and found that $w = 32$ and $p = 64$ result in a good tradeoff between running time and uplink goodput gain.

After determining the window size and sampling frequency, we need to design a policy on choosing the representative windows. Aggarwal et al. [17] propose a policy called SAMPLEBYTE, and show it outperforms other policies. We adopt SAMPLEBYTE in CacheQuery. Specifically, the receiver maintains a *marker* list of m byte values, where $1 \leq m \leq 256$. Whenever the receiver sees a new packet, it traverses through every byte of that packet, and compares its value against the markers' values. If there is a match at offset x, the receiver selects $[x, x+w-1]$ as the representative window, and skips $p/2$ bytes in order to comply with the sampling frequency. Different from the pre-computed static marker list used in Aggarwal et al. [17], CacheQuery dynamically computes the marker list based on the occurrence frequency of all byte values across the receiver cache. CacheQuery pushes the complexity of computing the marker list, along with other computations, to the powerful receiver. CacheQuery employs a marker list refresh threshold T_m packets, for statistically meaningful marker lists. The receiver updates the marker list once every $r_m = \max(f, T_m)$ packets, and transmits the list to the sender. We let $T_m = 1000$ if not otherwise specified.

Hashing representative windows. To facilitate fast lookup, we employ Jenkins Hash function [19] to compute a 32-bit hash code, referred to as *fingerprint*. The receiver maintains a hash table with fingerprint as keys, and <historical packet ID, offset> as values, where historical packet ID points to a specific packet in the cache. This is called the fingerprint table, which is sent to the sender whenever the receiver does a cache update. The sender uses this fingerprint table for common byte range lookups.

Locating matching byte range. For each packet, the sender uses the marker list to locate all representative windows in it. The sender then computes their fingerprints. Comparing against the fingerprint table, the sender finds the first matching window. It then expands the matching window to the left and right one byte after another, so as to maximize the matching byte range.

Encoding the matching byte range. The sender sends <historical packet ID, offset, length> instead of the byte range

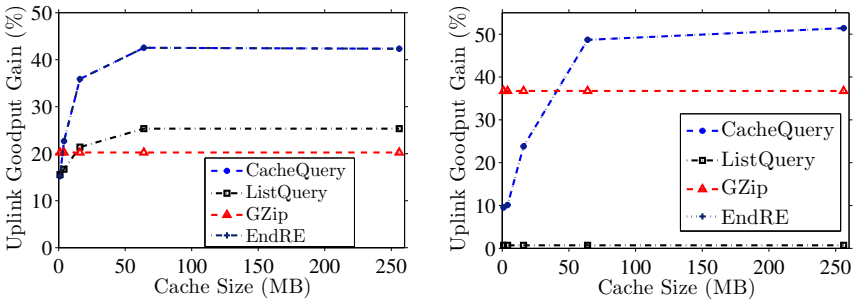


Fig. 3. Uplink goodput gain achieved by various protocols, sample results from: (a) T1 and (b) T5.

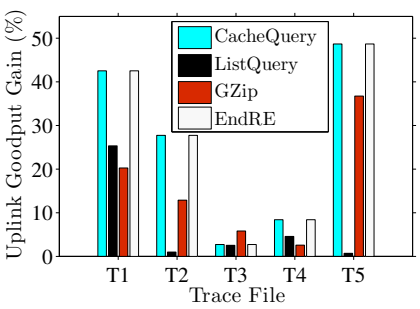


Fig. 4. Overall uplink goodput gain.

itself. The receiver uses the sender cache shadow to reconstruct the original byte range. Since the tuple is shorter than the raw byte range, CacheQuery may achieve high goodput gain.

V. Trace-Driven Simulations

A. Setup

We extend the event-driven simulator presented in Sec. III-A to support CacheQuery. We compare CacheQuery against ListQuery, because ListQuery outperforms all other asymmetric communication algorithms in terms of the uplink goodput gain, which is also shown in Sec. III-A. We also implement EndRE [17] and GZip [20] algorithms in the simulator for comparisons. The EndRE algorithm employs symmetric caches for sender-driven redundancy elimination, while GZip compresses the payload of each packet before sending it out. We use actual network traces collected in Sec. III-A (see Table I) to drive the simulator. We conduct both protocol-independent and per-protocol simulations. In the latter case, we split each network trace into multiple protocol-specific traces. We ignore the protocols with fewer than 1000 packets. We program the simulator to report the performance results after each round of simulation. We consider the following performance metrics: (i) uplink goodput gain in percentage, (ii) relative overhead, which is defined as the ratio of downlink traffic amount and the raw uplink traffic amount in percentage, and (iii) per-packet encoding and decoding time in msec.

For CacheQuery and EndRE, we let marker list length $m = 10$, update frequency $f = 1000$, receiver cache size $B_r = 64$, sender cache size $B_s = 16$, selection policy parameter $\beta = 0.1$, and trace file be T1, if not otherwise specified. We emphasize that, for fair comparisons, the sender and receiver cache sizes *include* all the storage overhead, in particular the fingerprint tables. For EndRE, the sender and receiver cache sizes must be identical, while the proposed CacheQuery allows the users to specify a smaller sender cache size. Various system parameters, including the update frequency, selection policy parameter, and sender cache size, are varied in the simulations to study their implications on system performance.

B. Results

Improved uplink goodput gain. We first compare the uplink goodput gain achieved by all considered algorithms. For fair comparisons, we let $B_r = B_s \in \{1, 4, 16, 64, 256, 512\}$ and $f = 1$ since EndRE only supports this configuration. We plot the sample results from T1 and T5 in Fig. 3, in which we skip $B_r = B_s = 512$ for brevity, as it leads to the similar results as $B_r = B_s = 256$. Note that, in this figure, CacheQuery achieves similar uplink goodput gain as EndRE; therefore their lines overlap with each other. Fig. 3(a) shows that CacheQuery outperforms GZip when $B_s = B_r \geq 4$. Moreover, CacheQuery always outperforms ListQuery: up to 1.54 times of uplink goodput gain is possible. Fig. 3(b) shows that CacheQuery significantly outperforms ListQuery: about 50 times uplink goodput gain improvement is observed. Fig. 4 present the overall results. This figure shows that CacheQuery constantly outperforms ListQuery, especially for T2 and T5, in which ListQuery leads to negligible uplink goodput gain. Figs. 3 and 4 clearly show that CacheQuery is one of the state-of-the-art asymmetric communication algorithms. Given that ListQuery and GZip lead to inferior performance, and EndRE dictates $B_s = B_r$ (thus is inflexible), we concentrate on the evaluation of CacheQuery in the rest of this section.

Implications of B_r and B_s. We vary $B_r \in \{1, 4, 16, 64, 256, 512\}$, and $B_s = \{1, 4, 16, 64\}$. We plot the uplink goodput gain in Fig. 5, in which we zoom into $B_r \in [0, 100]$. We make two observations. First, with $B_s \geq 4$, larger receiver cache leads to higher uplink goodput gain. Second, when sender cache size $B_s = 1$, larger receiver cache actually leads to *lower* goodput gain. We believe this is because larger B_r means more room for selecting representative windows, and thus scenarios with small B_s are more sensitive to the quality of window selection policy. Fig. 5 reveals the tight correlation between B_s and B_r: a joint decision on them need to be made for a good tradeoff between uplink goodput gain and resource consumption. Designing an algorithm to dynamically adjust B_s and B_r, along with other system parameters, is one of our future tasks.

Diversity of uplink goodput gain. We plot the protocol-independent and per-protocol uplink goodput gain in Fig. 6. This figure shows that the achieved gain of CacheQuery is quite diverse: 3–32% for protocol-independent case (Fig. 6(a)) and 2–57% for per-protocol case (Fig. 6(b)). In particular, SMTP (25), POP3S (995), and NFS (2049) achieve more than 40% gains. Fig. 6(b) also shows that the range of gains of a protocol with different traces could be large. For example, the HTTP protocol with different traces achieves very different

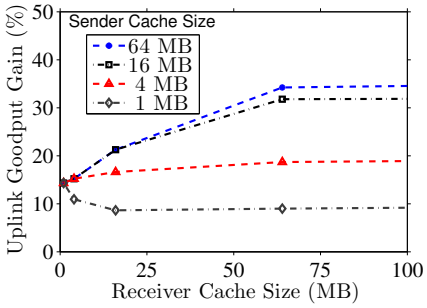


Fig. 5. Implications of B_r and B_s.

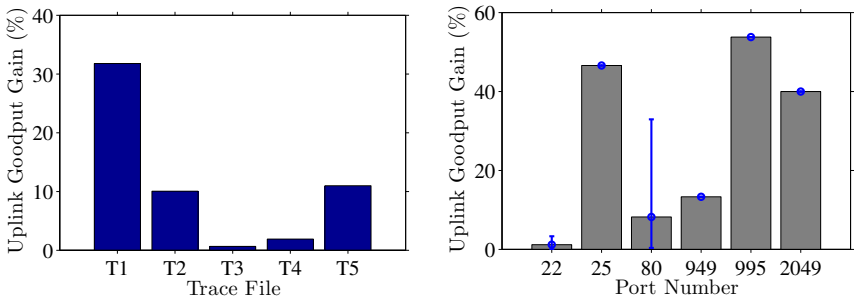


Fig. 6. Diverse goodput gains from: (a) various trace files and (b) per-protocol trace files.

gains. Hence, the uplink goodput gain of CacheQuery highly depends on the payload content.

Tradeoff between uplink goodput gain and relative overhead. While larger sender cache results in higher uplink goodput gain, it also leads to more downlink traffic. We plot the uplink goodput gain and relative overhead of CacheQuery and ListQuery with $B_s \in \{1, 4, 16, 32, 48, 64\}$ in Fig. 7. Fig. 7(a) shows that CacheQuery and ListQuery lead to similar gain when $B_s \leq 32$, and CacheQuery outperforms ListQuery when $B_s > 32$. Fig. 7(b) shows the relative overhead. It clearly illustrates that ListQuery suffers from huge relative overhead: it incurs up to 18450 times of downlink traffic amount, compared to the raw uplink traffic amount. In contrast, CacheQuery uses a small sender cache to *absorb* a huge portion of the downlink traffic, and incurs no more than 48 times of the downlink traffic amount (not visible due to the Y-axis scale), about 384 times lower than that of ListQuery.

Implications of system parameters β, m, and f. We vary $\beta \in \{0, 0.1, 0.25, 0.5, 0.75, 1\}$, $m = \{3, 5, 10, 20, 40\}$, and $f \in \{1000, 3000, 5000, 8000, 10000\}$. Fig. 8(a) shows the uplink goodput gain of CacheQuery with different β values, which reveals that $\beta \in [0.1, 0.25]$ achieves high uplink goodput gain for most traces. Fig. 8(b) illustrates the uplink goodput gain under different m values. It reveals that, in general, longer marker list results in higher uplink goodput gain. Nonetheless, we note that larger m value may lead to longer encoding and decoding time. We also find that smaller f leads to higher goodput gain but also incurs higher overhead (figures are given in [14] due to the space limitations).

Encoding and decoding time. We collect per-packet encoding and decoding time from a commodity 3.4 GHz Intel i7 Linux PC. The figures are given in [14] due to the space limitations. We set receiver cache size to $B_r = 64$ and vary sender cache size B_s. We find that the encoding overhead of the CacheQuery algorithm is less than 0.5 msec while the decoding overhead is less than 5.2 msec. Moreover, CacheQuery is scalable to the cache size, as its encoding/decoding time remains almost constant with different B_s.

VI. Extensions

A. Multi-Sender Scenarios

The previous discussion concentrates on single-sender scenarios. To leverage on packet redundancy across multiple senders, CacheQuery can be readily extended to multi-sender scenarios by maintaining a sender cache shadow on the receiver for each sender. We however acknowledge that maintaining separate sender cache shadows may consume more resources on the receiver; designing a more compact data structure is one of our future work.

B. Stream-Based Sender Cache: CacheQuery+

The encoding efficiency at the CacheQuery sender is constrained by its cache size B_s. When B_s is small, we propose CacheQuery+ to trade higher relative overhead for higher uplink goodput gain. CacheQuery+ is a variation of CacheQuery, which allows $f = B_s/B_r < 1$ when $B_r > B_s$. That is, the receiver sends *multiple* short cache updates to the sender for several buffered outgoing packets at the sender. In this way, the receiver can continuously *stream* the whole receiver cache to the sender even when $B_r \gg B_s$. Fig. 9 compares the results of CacheQuery+ with $B_s = 8$ and $B_r \in [1, 128]$, against EndRE with $B_s = B_r = 8$. Fig. 9(a) presents the sample result from trace T1, which shows that CacheQuery+ outperforms EndRE when $B_r \geq 64$, and by up to 10%. Fig. 9(b) reports the overall performance with $B_r = 128$, which illustrates that CacheQuery+ significantly outperforms EndRE with 3 out of 5 traces, and by up to 25%. We acknowledge that, compared to CacheQuery, CacheQuery+ may pay the expense of high downlink traffic amount; how to quantify and control it remains one of our future tasks.

VII. Conclusion and Future Work

In this paper, we proposed a practical asymmetric communication algorithm, called CacheQuery, for bandwidth and capability asymmetric channels. We collected real traffic traces from enterprise, home, and university servers. The extensive simulation results reveal the merits of CacheQuery. Compared to existing asymmetric communication algorithms, CacheQuery successfully improves the uplink goodput gain, up to 50 times of increase, while incurring small downlink traffic overhead, up to 384 times of reduction are observed. Compared to the state-of-the-art redundancy elimination algorithms, CacheQuery/CacheQuery+ leverages the idling downlink bandwidth and receiver capability for higher uplink goodput gain: up to 25% higher uplink goodput gain is possible. Moreover, CacheQuery shifts computational and storage complexities

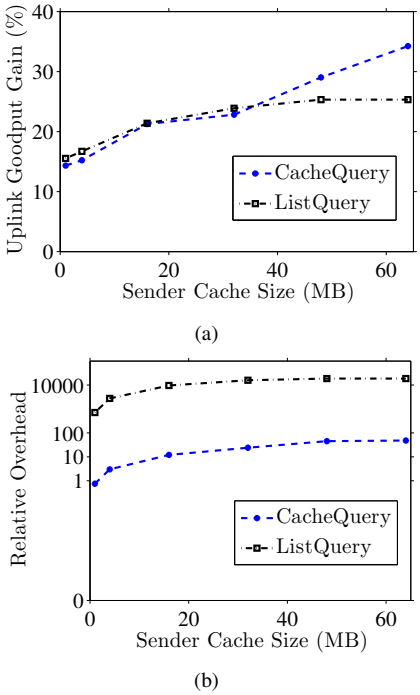


Fig. 7. Tradeoff between: (a) uplink goodput gain and (b) relative overhead.

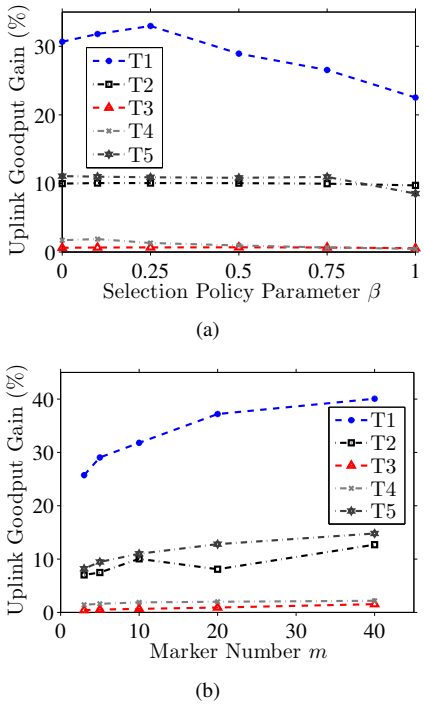


Fig. 8. Implications of the system parameters: (a) β and (b) m.

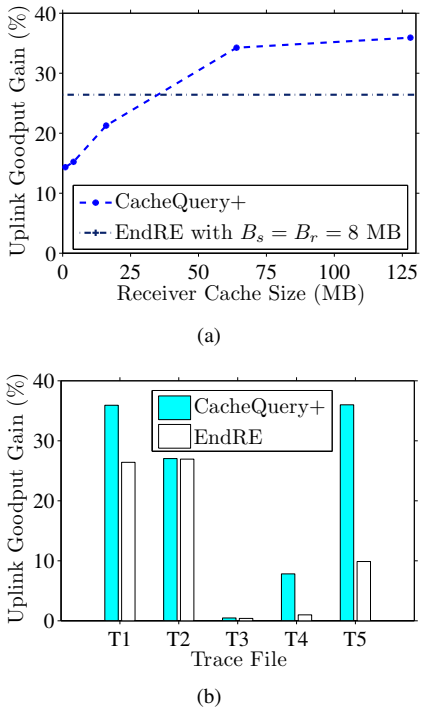


Fig. 9. Uplink goodput gain results from CacheQuery+ and EndRE: (a) sample results from T1 and (b) overall results with $B_r = 128$ MB.

from the sender to receiver, and thus is suitable to asymmetric communications.

We acknowledge that although our simulation results are very encouraging, our proposed algorithm might not work that well for certain types of traffic. To better understand its potential limitations, we are conducting more experiments using a large set of network traces collected from a high-speed network [21], which generates more than 4 TB data everyday.

REFERENCES

[1] B. Aslam, P. Wang, and C. Zou, "Pervasive Internet access by vehicles through satellite receive-only terminals," in *Proc. of IEEE International Conference on Computer Communications and Networks (ICCCN'09)*, San Francisco, CA, August 2009.

[2] H. Balakrishnan and V. Padmanabhan, "How network asymmetry affects TCP," *IEEE Communications Magazine*, vol. 39, no. 4, pp. 60–67, April 2001.

[3] M. Dischinger, A. Haeberlen, K. Gummadi, and S. Saroiu, "Characterizing residential broadband networks," in *Proc. of ACM SIGCOMM Internet Measurement Workshop (IMC'07)*, San Diego, CA, October 2007, pp. 43–56.

[4] R. Wang, T. Taleb, A. Jamalipour, and B. Sun, "Protocols for reliable data transport in space Internet," *IEEE Communications Surveys and Tutorials*, vol. 11, no. 2, pp. 21–32, Second Quarter 2009.

[5] M. Mathur, "Elucidation of upcoming traffic problems in cloud computing," *Recent Trends in Networks and Communications: Communications in Computer and Information Science*, vol. 90, no. 1, pp. 68–71, July 2010.

[6] L. Atzori, A. Iera, and G. Morabito, "The Internet of things: A survey," *Computer Networks*, vol. 54, no. 15, pp. 2787–2805, October 2010.

[7] M. Zorzi, A. Gluhak, S. Lange, and A. Bassi, "From today's INTRAnet of things to a future INTERnet of things: A wireless- and mobility-related view," *IEEE Wireless Communication Magazine*, vol. 17, no. 6, pp. 44–51, December 2010.

[8] M. Adler and B. Maggs, "Protocols for asymmetric communication channels," *Journal of Computer and System Sciences*, vol. 63, no. 4, pp. 573–596, December 2001.

[9] P. Bose, D. Krizanc, S. Langerman, and P. Morin, "Asymmetric communication protocols via hotlink assignments," *Theory of Computing Systems*, vol. 36, no. 6, pp. 655–661, November 2003.

[10] G. Mazzini, "Asymmetric channel cooperative compression," *IEEE Communications Letters*, vol. 12, no. 4, pp. 328–330, April 2008.

[11] T. Gagie, "Dynamic asymmetric communication," in *Proc. of International Colloquium on Structural Information and Communication Complexity (SIROCCO'06)*, Chester, UK, July 2006, pp. 310–318.

[12] ——, "Dynamic asymmetric communication," *Information Processing Letters*, vol. 108, no. 6, pp. 352–355, November 2008.

[13] C. Trang, X. Huang, and C. Hsu, "Pushing uplink goodput of an asymmetric access network beyond its uplink bandwidth," in *Proc. of IEEE International Conference on Communications (ICC'12)*, Ottawa, Canada, June 2012.

[14] Y. Li, C. Trang, X. Huang, C. Hsu, and P. Lin, "A practical asymmetric communication algorithm," Tech. Rep., March 2012, http://nmsl.cs.nthu.edu.tw/dropbox/CacheQueryTR.pdf.

[15] D. Munteanu and C. Williamson, "An FPGA-based network processor for IP packet compression," in *Proc. of SCS International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS'05)*, Philadelphia, PA, July 2005, pp. 599–608.

[16] C. Tye and D. Fairhurst, "A review of IP packet compression techniques," in *Proc. of Annual PostGraduate Symposium on the Convergence of Telecommunications, Networking and Broadcasting (PGNet'03)*, Liverpool, UK, June 2003.

[17] B. Aggarwal, A. Akella, A. Anand, A. Balachandran, P. Chitnis, C. Muthukrishnan, R. Ramjee, and G. Varghese, "EndRE: An end-system redundancy elimination service for enterprises," in *Proc. of USENIX Conference on Networked Systems Design and Implementation (NSDI'10)*, San Jose, CA, June 2010, pp. 419–432.

[18] E. Zohar, I. Cidon, and O. Mokryn, "The power of prediction: Cloud bandwidth and cost reduction," in *Proc. of ACM SIGCOMM'11*, Toronto, Canada, August 2011, pp. 86–97.

[19] B. Jenkins, "Algorithm alley: Hash functions," *Dr. Dobb's Journal*, September 1997.

[20] "zlib home page," April 2010, http://zlib.net/.

[21] Y.-D. Lin, I.-W. Chen, P.-C. Lin, C.-S. Chen, and C.-H. Hsu, "On campus beta site: architecture designs, operational experience, and top product defects," *IEEE Communications Magazine*, vol. 48, no. 12, pp. 83–91, December 2010.